

Cours de Python



<http://www.dsimb.inserm.fr/~fuchs/python/>

Patrick Fuchs et Pierre Poulain

prenom [dot] nom [at] univ-paris-diderot [dot] fr

version du 8 mars 2010

DSIMB,
Inserm UMR_S 665 et Université Paris Diderot-Paris 7,
Institut Nationale de la Transfusion Sanguine (INTS),
6 rue Alexandre Cabanel,
75015 Paris, France

Bienvenue au cours de Python !

Ce cours de Python s'adresse essentiellement aux débutants issus des filières de biologie / biochimie, et plus spécialement aux étudiants des masters BI (Biologie Informatique), BC2T (Biochimie, Cellules, Cibles Thérapeutiques) de l'[Université Paris Diderot - Paris 7](#). Ce cours est basé sur les versions 2.x de Python, et il ne sera pour l'instant pas compatible avec les nouvelles versions 3.0 et 3.1. Néanmoins, nous essaierons de mettre rapidement à jour le cours de manière à traiter la compatibilité entre les deux générations. Si vous relevez des erreurs en suivant ce cours, merci de nous les signaler.

Le cours est disponible en version [HTML](#) et [PDF](#).

Remerciements

Un grand merci à [Sander](#) du [CMBI](#) de Nijmegen pour la [première version](#) de ce cours.

Merci également aux contributeurs, occasionnels ou réguliers : Jennifer Becq, Virginie Martiny, Romain Laurent, Benoist Laurent, Benjamin Boyer, Hubert Santuz.

Enfin, merci à vous tous, les curieux de Python, qui avez été nombreux à nous envoyer des retours sur ce cours, à nous suggérer des corrections, à nous signaler des coquilles.

De nombreuses personnes nous ont aussi demandé les corrections des exercices, nous ne les mettons pas sur le site afin d'éviter la tentation de les regarder trop vite. Mais vous pouvez nous écrire et nous vous les enverrons.

Table des matières

1	Introduction	5
1.1	Avant de commencer	5
1.2	Premier contact avec Python sous Linux	5
1.3	Premier programme Python	6
2	Variables et Ecriture	7
2.1	Les Variables	7
2.2	Opérations sur les variables	7
2.3	Écriture	8
2.4	Écriture formatée	9
2.5	Exercices	11
3	Listes	12
3.1	Définition	12
3.2	Utilisation	12
3.3	Indiçage négatif et tranches	13
3.4	Les fonctions range et len	14
3.5	Listes de listes	15
3.6	Exercices	15
4	Les boucles et comparaisons	17
4.1	Les boucles for	17
4.2	Comparaisons	18
4.3	Les boucles while	19
4.4	Exercices	20
5	Les tests	21
5.1	Définition	21
5.2	Tests à plusieurs cas	21
5.3	Tests multiples	22
5.4	Exercices	23
6	Les fichiers	25
6.1	Lecture dans un fichier	25
6.2	Écriture dans un fichier	27
6.3	Exercices	27
7	Les modules	29
7.1	Définition	29
7.2	Importation de modules	29
7.3	Obtenir de l'aide sur les modules importés	30
7.4	Modules courants	31
7.5	Exercices	31
8	Les chaînes de caractères	33
8.1	Préambule	33
8.2	Chaînes de caractères et listes	33
8.3	Caractères spéciaux	33
8.4	Les méthodes associées aux objets de type string	34
8.5	Conversion de types	35
8.6	Exercices	36

9 Plus sur les listes	37
9.1 Propriétés des listes	37
9.2 Références partagées	38
9.3 Exercices	38
10 Les dictionnaires et les tuples	40
10.1 Les Dictionnaires	40
10.2 Les Tuples	41
10.3 Exercices	42
11 Les fonctions	43
11.1 Définition	43
11.2 Passage d'arguments	44
11.3 Portée des variables	45
11.4 Portée des listes	46
11.5 La règle LGI	47
11.6 Exercices	48
12 Les modules sys et re	49
12.1 Le module sys : passage d'arguments	49
12.2 Le module re : expressions régulières	49
12.3 Exercices	51
13 Le module pickle	53
13.1 Codage des données	53
13.2 Décodage des données	54
13.3 Exercices	54
14 Gestion des entrées et des erreurs	55
15 Création de ses propres modules	56
15.1 Création	56
15.2 Exercices	57

1 Introduction

1.1 Avant de commencer

Avant de commencer, voici quelques indications générales qui pourront vous servir pour ce cours.

Familiarisez-vous avec le site www.python.org. Il contient énormément d'informations et de liens sur Python et vous permet en outre de le télécharger pour différentes plateformes (Linux, Mac, Windows...). La page d'[index des modules](#) est particulièrement utile.

Si vous voulez aller plus loin avec Python, Gérard Swinnen a écrit un très bon [livre](#), téléchargeable gratuitement (les éditions O'Reilly proposent également la [version anglaise](#) de cet ouvrage).

L'apprentissage d'un langage informatique comme python va nécessiter d'écrire des lignes de codes à l'aide d'un éditeur de texte. Si vous êtes débutant, je vous conseille vivement d'utiliser *gedit* ou *nedit*, qui est les plus proches des éditeurs que l'on peut trouver sous Windows (*notepad*). Des éditeurs comme *emacs* et *vi* sont très puissants mais nécessitent un apprentissage particulier.

1.2 Premier contact avec Python sous Linux

Python est un langage interprété, c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée par l'ordinateur. Pour vous en rendre mieux compte, lancez la commande :

```
python
```

Celle-ci va démarrer l'**interpréteur Python**. Vous devriez obtenir quelque chose de ce style (où `[fuchs@opera ~]$` représente l'invite de votre shell) :

```
[fuchs@opera ~]$ python
Python 2.5.1 (r251:54863, Jul 10 2008, 17:25:56)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le triple chevron `>>>` est l'invite de Python (*prompt* en anglais), ce qui signifie que Python attend une commande. Tapez par exemple la commande suivante :

```
>>> print "Hello world !"
Hello world !
>>>
```

Python a exécuté la commande directement et a affiché `Hello world !`. Il attend ensuite votre prochaine instruction en affichant l'invite de l'interpréteur Python (`>>>`).

Vous pouvez refaire un nouvel essai en vous servant cette fois de l'interpréteur comme d'une machine à calculer.

```
>>> 1 + 1
2
>>>
```

À ce stade, vous pouvez entrer une autre commande ou bien quitter l'interpréteur à l'aide des touches `Ctrl-D`. Finalement l'interpréteur Python est un système interactif dans lequel vous

pouvez entrer des commandes, que Python exécutera sous vos yeux (au moment où vous validerez la commande en tapant sur `Entrée`).

Il existe de nombreux autres langages interprétés tels que `perl`, `R`... Le gros avantage est que l'on peut directement tester une commande à l'aide de l'interpréteur, ce qui est très utile pour déboguer (c'est-à-dire corriger les éventuelles erreurs d'un programme). Gardez bien en mémoire cette propriété de Python qui pourra parfois vous faire gagner un temps précieux !

1.3 Premier programme Python

Bien sûr, l'interpréteur présente vite des limites dès lors que l'on veut exécuter une suite d'instructions plus complexe. Comme tout langage informatique, on peut enregistrer ces instructions dans un fichier, que l'on appelle communément un script Python.

Pour reprendre l'exemple précédent, ouvrez un éditeur de texte (par exemple `gedit` ou `nedit`) et tapez le code suivant.

```
print 'Hello World !'
```

Ensuite enregistrez votre fichier sous le nom `test.py`, puis quittez l'éditeur de texte. L'extension standard des scripts Python est `.py`. Pour exécuter votre script, vous avez deux moyens :

1. Donner le nom de votre script comme argument à la commande Python :

```
[fuchs@opera ~]$ python test.py
Hello World !
[fuchs@opera ~]$
```

2. Rendre votre fichier exécutable ; pour cela deux opérations sont nécessaires :

- Indiquer au shell la localisation de l'interpréteur Python en indiquant dans la première ligne du script :

```
#!/usr/bin/env python
```

Dans notre exemple, le script `test.py` contient alors le texte suivant :

```
#!/usr/bin/env python
```

```
print 'Hello World !'
```

- Rendre votre script python exécutable en tapant :

```
chmod 755 test.py
```

Pour exécuter votre script, tapez son nom précédé des deux caractères `./` (afin de préciser au shell où se trouve votre script) :

```
[fuchs@opera ~]$ ./test.py
Hello World !
[fuchs@opera ~]$
```

2 Variables et Ecriture

2.1 Les Variables

Une **variable** est une zone de la mémoire dans laquelle on stocke une **valeur**. Aux yeux du programmeur, cette variable est définie par un **nom**, alors que pour l'ordinateur, il s'agit en fait d'une adresse (i.e. une zone particulière de la mémoire).

Le **type** d'une variable correspond à la nature de celle-ci. Les 3 types principaux dont nous aurons besoin sont les entiers, les réels et les chaînes de caractères. Bien sûr, il existe de nombreux autres types (e.g. pour les nombres complexes), c'est d'ailleurs un des gros avantages de Python (si vous n'êtes pas effrayés, vous pouvez vous en rendre compte [ici](#)).

En Python, la **déclaration** d'une variable et son **initialisation** (c.à.d. la première valeur que l'on va stocker dedans) se fait en même temps. Pour vous en convaincre, regardez puis testez l'exemple suivant après avoir lancé l'interpréteur :

```
>>> x = 2
>>> x
2
>>>
```

Dans cet exemple, nous avons déclaré, puis initialisé la variable `x` avec la valeur 2. L'interpréteur nous a ensuite permis de regarder le contenu de la variable juste en tapant son nom (retenez ceci pour le *debugging* dans l'interpréteur Python). Dans notre exemple, nous avons stocké un nombre entier (*int*) dans la variable `x`, mais il est tout à fait possible de stocker des nombres réels (*float*) ou des chaînes de caractères (*string*) :

```
>>> x = 3.14
>>> x
3.1400000000000001
>>> x = "Bonjour !"
>>> x
'Bonjour !'
>>> x = 'Bonjour !'
>>> x
'Bonjour !'
>>>
```

Vous remarquez que Python reconnaît certains types de variable automatiquement (entier, réel). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (simples ou doubles) afin d'indiquer à Python le début et la fin de cette chaîne.

2.2 Opérations sur les variables

Les 4 opérations de base se font de manière simple sur les types numériques (nombres entiers et réels) :

```
>>> x = 45
>>> x + 2
47
>>> y = 2.5
>>> x + y
47.5
>>> (x * 10) / y
180.0
>>>
```

Remarquez toutefois que si vous mélangez les types entiers et réels, le résultat vous est renvoyé comme un réel (car ce type est plus général).

Pour les chaînes de caractères, deux opérations sont possibles :

```
>>> chaine = "Salut"
>>> chaine
'Salut'
>>> chaine + " Python"
'Salut Python'
>>> chaine * 3
'SalutSalutSalut'
>>>
```

L'opérateur d'addition `+` permet de concaténer (rassembler) deux chaînes de caractères, et l'opérateur de multiplication `*` permet de dupliquer `n` fois une chaîne.

Attention à ne pas mélanger les types car vous obtiendriez un message d'erreur :

```
>>> 'toto' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Notez que Python essaie de vous donner le maximum d'indices dans son message d'erreur. Dans l'exemple précédent, il vous indique que vous ne pouvez pas mélanger des objets de type `str` (*string*, donc chaînes de caractères) avec des objets de type `int` (dont des entiers), ce qui est assez logique.

Si vous ne vous souvenez plus du type d'une variable, vous pouvez utiliser la fonction `type` qui va vous le rappeler.

```
>>> x = 2
>>> type(x)
<type 'int'>
>>> x = 2.0
>>> type(x)
<type 'float'>
>>> x = '2'
>>> type(x)
<type 'str'>
>>>
```

Faites bien attention, pour Python, la valeur `2` (nombre entier) est différente de `2.0` (nombre réel), de même que `2` (nombre entier) est différent de `'2'` (chaîne de caractères).

2.3 Écriture

Nous avons déjà vu au paragraphe précédent la fonction `print` qui permet d'écrire une chaîne de caractères ; elle permet en outre d'écrire le contenu d'une ou plusieurs variables :

```
>>> x = 32
>>> nom = 'John'
>>> print nom , ' a ' , x , ' ans'
John a 32 ans
```

Python a donc écrit la phrase en remplaçant les variables par leur contenu. Vous pouvez noter également que pour écrire plusieurs blocs de texte, nous avons utilisé le séparateur `,` avec la commande `print`. En regardant de plus près, vous vous apercevrez que Python a automatiquement ajouté un espace à chaque fois que l'on utilisait le séparateur `,`. Par conséquent, si vous voulez mettre un seul espace entre chaque bloc, vous pouvez retirer ceux de vos chaînes de caractères :

```
>>> print nom , 'a' , x , 'ans'
John a 32 ans
>>>
```

Pour imprimer deux chaînes de caractères l'une à côté de l'autre sans espace, vous devrez les concaténer :

```
>>> codon1 = 'ATG'
>>> codon2 = 'GCC'
>>> print codon1,codon2
ATG GCC
>>> print codon1 + codon2
ATGGCC
>>>
```

2.4 Écriture formatée

Imaginez que vous vouliez calculer puis afficher la proportion de GC d'un génome. Notez que la proportion de GC s'obtient comme la somme des bases G et C divisée par le nombre total de bases. Sachant que l'on a 4502 bases C, 2567 bases G pour un total de 15003 bases, vous pourriez faire comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
>>> propGC = (4502.0 + 2567)/15003
>>> print "La proportion de GC est",propGC
La proportion de GC est 0.47117243218
>>>
```

Remarquez que si vous aviez fait le calcul avec $(4502 + 2567)/15003$, vous auriez obtenu 0 car tous les nombres sont des entiers et le résultat aurait été, lui aussi, un entier. L'introduction de 4502.0 qui est un réel résoud le problème, puisque le calcul se fait alors sur des réels.

Néanmoins, le résultat obtenu présente trop de décimales (onze dans le cas présent). Pour pouvoir écrire le résultat plus lisiblement, vous pouvez utiliser l'opérateur de formatage `%`. Dans votre cas, vous voulez formater un réel pour l'afficher avec 2 puis 3 décimales :

```
>>> print "La proportion de GC est %.2f" % propGC
Le proportion de GC est 0.47
>>> print "La proportion de GC est %.3f" % propGC
La proportion de GC est 0.471
>>>
```

Le signe `%` est appelé une première fois (`%.2f`) pour

1. désigner l'endroit où sera placée la variable dans la chaîne de caractères ;
2. préciser le type de la variable à formater, ici `f` pour *float* donc pour un réel ;

3. préciser le formatage voulu, ici `.2` soit deux chiffres après la virgule.

Le signe `%` est rappelé une seconde fois (`% propGC`) pour indiquer les variables à formater.

Vous pouvez aussi formatez des entiers

```
>>> nbG = 4502
>>> print "Le génome de cet exemple contient %i G" % nbG
Le génome de cet exemple contient 4502 G
>>>
```

ou mettre plusieurs nombres dans une même chaîne de caractères.

```
>>> nbG = 4502
>>> nbC = 2567
>>> print "Ce génome contient %i G et %i C, soit une proportion de GC de %.2f" \
... % (nbG,nbC,propGC)
Ce génome contient 4502 G et 2567 C, soit une proportion de GC de 0.47
>>>
```

Remarque :

1. Dans l'exemple précédent, nous avons utilisé le symbole `%` pour formater des variables. Si vous avez besoin d'écrire le symbole pourcentage `%` sans qu'il soit confondu avec celui servant pour l'écriture formatée, il suffit de le doubler (comme en C). Toutefois, si l'écriture formatée n'est pas utilisée dans la même chaîne de caractères où vous voulez utiliser le symbole pourcentage, cette opération n'est pas nécessaire. Par exemple :

```
>>> nbG = 4502
>>> nbC = 2567
>>> percGC = propGC * 100
>>> print "Ce génome contient %i G et %i C, soit un %%GC de %.2f" \
... % (nbG,nbC,percGC) , "%"
Ce génome contient 4502 G et 2567 C, soit un %GC de 47.12 %
>>>
```

2. Le signe `\` en fin de ligne permet de poursuivre la commande sur la ligne suivante. Cette syntaxe est pratique lorsque vous voulez taper une commande longue.

Enfin, il est possible de préciser sur combien de caractères vous voulez qu'un résultat soit écrit. Dans la portion de code suivant, le caractère `;` sert de séparateur entre les instructions sur une même ligne :

```
>>> print 10 ; print 100 ; print 1000
10
100
1000
>>> print "%4i" % 10 ; print "%4i" % 100 ; print "%4i" % 1000
  10
 100
1000
>>>
```

Ceci est également possible sur les chaînes de caractères :

```
>>> print "atom HN" ; print "atom HDE1"
atom HN
```

```
atom HDE1
>>> print "atom %4s" % "HN" ; print "atom %4s" % "HDE1"
atom   HN
atom  HDE1
>>>
```

Vous voyez tout de suite l'énorme avantage de l'écriture formatée. Cela permet d'écrire en colonnes parfaitement alignées. Nous verrons que ceci est très pratique si l'on veut écrire les coordonnées d'une molécule au format PDB.

2.5 Exercices

1. Ouvrez l'interpréteur Python et taper `1+1`. Que se passe-t-il ? Étonnant non ? Écrivez la même chose dans un script `test.py`. Exécuter ce script en tapant `python test.py` dans un shell. Que se passe-t-il ? Pourquoi ?
2. Calculez le pourcentage de GC avec le code suivant

```
percGC = ((4502 + 2567)/15003)*100
```

 puis affichez le résultat. Que se passe-t-il ? Comment expliquez-vous ce résultat ?
3. Calculez les pourcentages de G et de C du génome du dahu, puis écrivez les avec 2 décimales de précision.
4. Générez une chaîne de caractères représentant un oligonucléotide polyA (AAAA...) de 20 bases de longueur, sans les taper littéralement.
5. Suivant le modèle du dessus, générez en une ligne de code un polyA de 20 bases suivi d'un polyGC régulier (GCGCGC...) de 40 bases.
6. En utilisant l'écriture formatée, affichez en une seule ligne les variables `a`, `b` et `c` dont les valeurs sont respectivement `"salut"`, `102` et `10.318`.

Conseil : utilisez l'interpréteur Python pour les exercices 2 à 5 !

3 Listes

3.1 Définition

Une **liste** est une structure de données qui contient une série de valeurs. Ces valeurs n'étant pas forcément du même type, ceci confère une grande flexibilité à ces listes. Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**. En voici quelques exemples :

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille = [5.0, 1.0, 0.7, 2.0]
>>> mixte = ['girafe', 5.0, 'dahu', 2]
>>> animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> print animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille
[5.0, 1.0, 0.7, 2.0]
>>> mixte
['girafe', 5.0, 'dahu', 2]
>>>
```

Lorsque l'on rappelle une liste, Python la restitue telle qu'elle a été saisie (aux espaces près).

3.2 Utilisation

Un des gros avantages d'une liste est que l'on peut rappeler ses éléments par leur numéro de position dans celle-ci ; on appelle ce numéro **indice** de la liste. Une chose est cependant à préciser et à **retenir** : l'indice d'une liste de n éléments commence à 0 et se termine à $n-1$. Voyez l'exemple suivant :

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> animaux[0]
'girafe'
>>> animaux[1]
'hippopotame'
>>> animaux[3]
'dahu'
>>>
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
>>> animaux[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

N'oubliez pas ceci, ou vous risqueriez d'obtenir des bugs inattendus !

Tout comme les chaînes de caractères les listes supportent l'opérateur + de concaténation, ainsi que * pour la duplication :

```
>>> animaux = ['aigle', 'ecureuil']
>>> animaux + animaux
['aigle', 'ecureuil', 'aigle', 'ecureuil']
>>> animaux * 3
['aigle', 'ecureuil', 'aigle', 'ecureuil', 'aigle', 'ecureuil']
>>>
```

On peut également considérer une chaîne de caractères comme une liste (de caractères) :

```
>>> animal = "hippopotame"
>>> animal
'hippopotame'
>>> animal[0]
'h'
>>> animal[4]
'o'
>>> animal[10]
'e'
>>>
```

3.3 Indiciage négatif et tranches

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

liste	:	['girafe', 'hippopotame', 'singé', 'vache']
index positif :		0 1 2 3
index négatif :		-4 -3 -2 -1

ou encore :

liste	:	['A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', ..., 'T', 'V', 'W', 'Y']
index positif :		0 1 2 3 4 5 6 7 8 9 10 ...16 17 18 19
index négatif :		-20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10 ...-4 -3 -2 -1

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez appeler le dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de la liste.

```
>>> animaux = ['girafe', 'hippopotame', 'singé', 'vache']
>>> animaux[-4]
'girafe'
>>> animaux[-2]
'singé'
>>> animaux[-1]
'vache'
>>>
```

Un autre avantage des listes est la possibilité de sélectionner une partie en utilisant un indiciage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du *m*ème au *n*ème (de l'élément *m* inclus à l'élément *n+1* non inclus). On dit alors qu'on récupère une **tranche** de la liste, par exemple :

```
>>> animaux = ['girafe', 'hippopotame', 'singé', 'vache', 'ornithorynque']
>>> animaux[0:2]
['girafe', 'hippopotame']
```

```

>>> animaux[0:3]
['girafe', 'hippopotame', 'singe']
>>> animaux[0:]
['girafe', 'hippopotame', 'singe', 'vache', 'ornithorynque']
>>> animaux[: ]
['girafe', 'hippopotame', 'singe', 'vache', 'ornithorynque']
>>> animaux[1:]
['hippopotame', 'singe', 'vache', 'ornithorynque']
>>> animaux[1:-1]
['hippopotame', 'singe', 'vache']
>>>

```

On peut remarquer que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole `:`, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

Dans les versions récentes de Python, on peut aussi préciser le pas en ajoutant un `:` supplémentaire et en indiquant le pas par un entier :

```

>>> x=range(10)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
>>>

```

Finalement, on voit que l'accès aux contenu des listes avec des crochets fonctionne sur le modèle `liste[debut:fin:pas]`.

3.4 Les fonctions range et len

L'instruction `range` vous permet de créer des listes d'entiers (et d'entiers uniquement) de manière simple et rapide. Voyez plutôt :

```

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,5)
[0, 1, 2, 3, 4]
>>> range(15,21)
[15, 16, 17, 18, 19, 20]
>>> range(0,1000,100)
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
>>> range(0,-10,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>>

```

L'instruction `range` fonctionne sur le modèle : `range([debut,] fin[, pas])` (les arguments entre crochets sont optionnels).

L'instruction `len` vous permet de connaître la longueur d'une liste, ce qui parfois est bien pratique lorsqu'on lit un fichier par exemple, et que l'on ne connaît pas *a priori* la longueur des lignes. En voici un exemple :

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu', 'ornithorynque']
>>> len(animaux)
5
>>> len(range(10))
10
>>>
```

3.5 Listes de listes

Pour finir, sachez qu'il est tout-à-fait possible de construire des listes de listes. Cette fonctionnalité peut être parfois très pratique (nous en verrons un exemple approfondi lorsque nous aborderons les tests). Par exemple :

```
>>> aa1 = ['Ala', 'alpha']
>>> aa2 = ['Lys', 'coil']
>>> aa3 = ['Ala', 'alpha']
>>> aa4 = ['Lys', 'alpha']
>>> peptide = [aa1, aa2, aa3, aa4]
>>> peptide
[['Ala', 'alpha'], ['Lys', 'coil'], ['Ala', 'alpha'], ['Lys', 'alpha']]
>>>
```

Dans cet exemple, chaque sous-liste contient un acide aminé et sa structure secondaire. Pour accéder à un élément de la sous-liste, on utilise un double indiciage :

```
>>> peptide[1]
['Lys', 'coil']
>>> peptide[1][0]
'Lys'
>>> peptide[1][1]
'coil'
>>>
```

On verra un peu plus loin qu'il existe en Python les dictionnaires qui sont très pratiques pour faire ce genre de choses.

3.6 Exercices

1. Constituez une liste `semaine` contenant les 7 jours de la semaine. À partir de cette liste, comment récupérez-vous seulement les 5 premiers jours de la semaine d'une part, et ceux du week-end d'autre part (*utilisez pour cela l'indiciage*)? Cherchez un autre moyen pour arriver au même résultat (*en utilisant un autre indiciage*).
2. Trouvez 2 manières pour accéder au dernier jour de la semaine.
3. Inversez les jours de la semaine en une commande.
4. Créez 4 listes `hiver`, `printemps`, `ete`, `automne` contenant les mois correspondant à ces saisons. Créez ensuite une liste `saisons` contenant les sous-listes `hiver`, `printemps`, `ete`, `automne`. Prévoyez ce que valent les variables suivantes, puis vérifiez le dans l'interpréteur :
- `saisons[2]`

- `saisons[1][0]`
- `saisons[1:2]`
- `saisons[:][1]`

Comment expliquez-vous ce dernier résultat ?

5. Affichez la table de 9 en une seule commande et de 2 manières différentes.
6. Avec Python, répondez à la question suivante en une seule commande. Combien y a-t-il de nombres pairs dans l'intervalle [2 , 10000] inclus ?

Conseil : utilisez l'interpréteur Python !

4 Les boucles et comparaisons

4.1 Les boucles for

Imaginez que vous ayez une liste de 10 éléments dont vous voulez écrire le contenu. Dans l'état actuel de vos connaissances, il faudrait taper quelque chose du style :

```
peptide = ['PRO', 'GLY', 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLN', 'GLU', 'HIS']
print peptide[0]
print peptide[1]
print peptide[2]
print peptide[3]
print peptide[4]
print peptide[5]
print peptide[6]
print peptide[7]
print peptide[8]
print peptide[9]
```

Si votre liste contient 10 éléments, ceci est encore faisable, mais imaginez qu'elle en contienne 100 voire 1000 ! Pour remédier à cela, il faut utiliser les boucles. Regardez l'exemple suivant :

```
>>> peptide = ['PRO', 'GLY', 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLN', 'GLU', 'HIS']
>>> for i in peptide:
...     print i
...
PRO
GLY
ALA
ARG
ASN
ASP
CYS
GLN
GLU
HIS
>>>
```

En fait, la variable `i` va prendre successivement les différentes valeurs de la liste `peptide` à chacune de ses itérations. D'ores et déjà je vous invite à remarquer avec attention l'**indentation** : vous voyez que l'instruction `print i` est légèrement décalée par rapport à l'instruction `for i in peptide` : outre une meilleure lisibilité, ceci est formellement requis en Python : **le corps de la boucle** (c-à-d toutes les instructions que l'on veut répéter) **doit être indenté** (d'un(e) ou plusieurs espace(s) ou tabulation(s)). Dans le cas contraire, Python vous renvoie un message d'erreur :

```
>>> for i in peptide:
... print i
    File "<stdin>", line 2
      print i
        ^
IndentationError: expected an indented block
>>>
```

Cette indentation correspond en fait aux caractères `{` et `}` du C. Un autre élément important est à préciser : si le corps de votre boucle contient plusieurs instructions, celles-ci doivent être

indentées de la même manière (e.g. si vous avez indenté la première instruction avec 2 espaces, vous devez faire de même avec la deuxième instruction, etc).

Remarquez également un autre aspect de la syntaxe, une instruction `for` doit **absolument** se terminer par le signe deux-points `:`.

Il est aussi possible d'utiliser une tranche d'une liste :

```
>>> peptide = ['PRO', 'GLY', 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLN', 'GLU', 'HIS']
>>> for i in peptide[3:7]:
...     print i
...
ARG
ASN
ASP
CYS
>>>
```

On a vu que les boucles `for` pouvaient utiliser une liste contenant des chaînes de caractères, mais elle peuvent tout aussi bien utiliser des listes numériques. En utilisant l'instruction `range` on peut facilement accéder à l'équivalent d'une commande C telle que

```
for (i = 0 ; i < 5 ; i++)
```

ce qui donne en Python :

```
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>>
```

Par conséquent, vous pouvez utiliser les boucles comme une itération sur des indices :

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'vache']
>>> for i in range(4):
...     print animaux[i]
...
girafe
hippopotame
singe
vache
>>>
```

Vous retrouvez ici la philosophie des boucles en C, mais retenez que les boucles `for` en Python peuvent utiliser directement les listes quel qu'en soit leur type.

4.2 Comparaisons

Avant de passer à une autre sorte de boucles (les boucles `while`), nous abordons tout de suite les **comparaisons**. Celles-ci seront reprises dans le chapitre des tests. Python est capable d'effectuer toute une série de comparaisons telles que :

syntaxe Python	signification
<code>==</code>	égal à
<code>!=</code>	différent de
<code>></code>	supérieur à
<code>>=</code>	supérieur ou égal à
<code><</code>	inférieur à
<code><=</code>	inférieur ou égal à

Observez l'exemple suivant sur des nombres entiers :

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
>>>
```

Python renvoie la valeur `True` si la comparaison est vraie, et `False` si elle est fausse. Faites bien attention à ne pas confondre l'opérateur d'affectation `=` qui donne une valeur à une variable et l'opérateur de comparaison `==` qui compare les valeurs de deux variables.

Vous pouvez également effectuer des comparaisons sur des chaînes de caractères :

```
>>> animal = "hippopotame"
>>> animal == "hippo"
False
>>> animal != "hippo"
True
>>> animal == 'hippopotame'
True
>>> len(animal) == 11
True
>>>
```

Dans le cas des chaîne de caractères, seuls les tests `==` et `!=` ont un sens.

4.3 Les boucles while

Une autre alternative à l'instruction `for` couramment utilisée en informatique est la boucle `while`. Le principe est simple : une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```
>>> i = 1
>>> while i <= 5:
...     print i
...     i = i + 1
...
1
2
3
4
5
>>>
```

Remarquez qu'il est encore une fois nécessaire d'indenter la série d'instructions. De plus, faites bien attention aux tests et à l'incrémation que vous utilisez car une erreur mène souvent à des boucles infinies (vous pouvez toujours stopper un script à l'aide des touches Ctrl-C).

4.4 Exercices

1. Soit la liste `['vache', 'souris', 'levure', 'bacterie']`. Affichez l'ensemble des éléments de cette liste (1 élément par ligne) de 3 manières différentes (2 avec `for` et une avec `while`).
2. Reprenez la liste avec les jours de la semaine (cf exercices sur les listes). Écrivez une série d'instructions affichant les jours de la semaine (en utiliser une boucle `for`), ainsi qu'une autre série d'instructions affichant les jours du week-end (en utilisant une boucle `while`).
3. Voici des notes d'étudiants `[5, 3, 2.5, 7, 0.5, 19]`. Remarquez qu'en Python le point est le séparateur décimal et que la virgule est le symbole séparant les éléments d'une liste. Calculez la moyenne de ces notes. Que pensez-vous de cette moyenne ?
4. En une seule ligne de Python, affichez les nombres de 1 à 10 sur une seule ligne.
5. Soit la liste `X` contenant les nombres entiers de 0 à 10. Calculez le produit des nombres consécutifs de `X` en utilisant une boucle.
6. Soit la chaîne de caractères `ACGTTAGCTAACGATCGGTAGCTGGT` représentant une séquence d'ADN. Initialisez la variable `seq` avec cette chaîne ainsi qu'une autre variable `liste` avec des nombres (pensez à `range`!). Remplissez les cases de `liste` avec les bases de `seq` à l'aide d'une boucle.

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un shell.

5 Les tests

5.1 Définition

Les **tests** sont un élément essentiel à tout langage informatique si on veut lui donner un peu de flexibilité. Ils permettent à l'ordinateur de prendre des décisions si telle ou telle condition est vraie ou fausse. Pour cela, Python utilise l'instruction `if` ainsi qu'une comparaison que nous avons abordée au chapitre précédent. En voici un exemple :

```
>>> x = 2
>>> if x == 2:
...     print 'Le test est vrai !'
...
Le test est vrai !
>>> x = 'hippopotame'
>>> if x == 'hippopotame':
...     print 'Le test est vrai !'
...
>>>
```

Plusieurs remarques concernant ces 2 exemples :

- Dans le premier exemple, le test étant vrai, l'instruction `print 'Le test est vrai !'` est exécutée, alors qu'elle ne l'est pas dans le deuxième.
- Les tests doivent forcément utiliser l'indentation comme les boucles `for` et `while`; celle-ci indique la portée des instructions à exécuter si le test est vrai.
- L'instruction `if` se termine comme les instructions `for` et `while` par le caractère `:`.

5.2 Tests à plusieurs cas

Parfois, il est pratique de tester si la condition est vraie ou si elle est fausse dans une même instruction `if`. Plutôt que d'utiliser 2 instructions `if`, on peut se servir de `if` et de `else` :

```
>>> x = 2
>>> if x == 2:
...     print 'Le test est vrai !'
... else:
...     print 'Le test est faux !'
...
Le test est vrai !
>>> x = 3
>>> if x == 2:
...     print 'Le test est vrai !'
... else:
...     print 'Le test est faux !'
...
Le test est faux !
>>>
```

Encore plus fort, on peut utiliser une série de tests dans la même instruction `if`, par exemple si on veut tester plusieurs valeurs d'une même variable. Par exemple, on se propose de tirer au sort un nombre entier, et on veut en déterminer la longueur en caractère(s) (dans le code suivant, nous utilisons l'instruction `random.randint(a, b)` qui renvoie un entier N avec $a \leq N \leq b$; l'instruction `import random` sera vue plus tard, admettez pour le moment qu'elle est nécessaire) :

```

>>> import random
>>> i = random.randint(0,1000)
>>> if i < 10:
...     print 'Le nombre i fait 1 caractere de large : ',i
... elif i < 100:
...     print 'Le nombre i fait 2 caracteres de large : ',i
... elif i < 1000:
...     print 'Le nombre i fait 3 caracteres de large : ',i
... elif i == 1000:
...     print 'Le nombre i fait 4 caracteres de large : ',i
...
Le nombre i fait 3 caracteres de large : 669
>>>

```

Dans cet exemple, Python teste la première condition, puis si et seulement si elle est fausse, il teste la deuxième et ainsi de suite... Le code correspondant à la première condition qui est vraie est exécuté puis Python sort du `if`.

Remarque : de nouveau, faites bien attention à l'indentation dans ces 2 derniers exemples ! Vous vous devez d'avoir une précision absolue à ce niveau là. Pour vous en persuader, exécutez ces 2 scripts dans l'interpréteur Python :

```

- Script 1 :
nombres = [5,6]
for i in nombres:
    if i <= 5:
        print 'La comparaison "i <= 5" est VRAIE'
        print 'parce-que i vaut',i
- Script 2 :
nombres = [5,6]
for i in nombres:
    if i <= 5:
        print 'La comparaison "i <= 5" est VRAIE'
        print 'parce-que i vaut',i

```

Comment expliquez-vous ce résultat ?

5.3 Tests multiples

Les tests multiples permettent de tester plusieurs conditions en même temps en utilisant des opérateurs booléens. Les 2 opérateurs les plus couramment utilisés sont le **OU** et le **ET**. Voici un petit rappel du mode de fonctionnement de ces opérateurs :

Condition 1	Opérateur	Condition 2	Résultat
Vrai	OU	Vrai	Vrai
Vrai	OU	Faux	Vrai
Faux	OU	Vrai	Vrai
Faux	OU	Faux	Faux
Vrai	ET	Vrai	Vrai
Vrai	ET	Faux	Faux
Faux	ET	Vrai	Faux
Faux	ET	Faux	Faux

En Python, on utilise le mot réservé `and` pour l'opérateur **ET**, et le mot réservé `or` pour l'opérateur **OU** (*attention à respecter la casse*). En voici un exemple d'utilisation :

```
>>> x = 2 ; y = 2
>>> if x == 2 and y == 2:
...     print "le test est vrai"
...
le test est vrai
>>>
```

Notez que l'on aurait pu obtenir le même résultat que dans cet exemple en utilisant 2 instructions `if` imbriquées :

```
>>> if x == 2:
...     if y == 2:
...         print "le test est vrai"
...
le test est vrai
>>>
```

Vous pouvez tester directement l'effet de ces opérateurs à l'aide des termes réservés `True` et `False` (*attention à respecter la casse*).

```
>>> True or False
True
>>>
```

On peut aussi utiliser l'opérateur logique de négation `not` qui inverse le résultat d'une condition :

```
>>> not True
False
>>> not False
True
>>>
```

5.4 Exercices

1. Reprenons notre liste `semaine` (cf exercice 1 sur les listes) ; en utilisant une boucle, écrivez chaque jour de la semaine ainsi que les messages suivants :
 - Au travail ! s'il s'agit du Lundi au Jeudi
 - Chouette c'est Vendredi ! s'il s'agit du Vendredi
 - Arggh, ce week-end je dois préparer un exposé pour Lundi ! s'il s'agit du week-end.(Les messages ne sont que des suggestions, vous pouvez laisser aller votre imagination !)
2. En reprenant notre exemple sur la séquence d'ADN (cf exercice 6 sur les boucles), écrire un script transformant celle-ci en sa séquence complémentaire.
3. Les angles dièdres ϕ/ψ d'une hélice α parfaite ont une valeur de -57 degrés et -47 degrés respectivement. Bien sûr, il est très rare que l'on trouve ces valeurs parfaites dans une protéine, par conséquent il est couramment accepté de tolérer une déviation de ± 30 degrés sur celles-ci. Ci-dessous vous avez une liste de listes contenant les valeurs de ϕ/ψ de la première hélice de la chaîne 1tfe. A l'aide de cette liste, écrire un programme qui teste pour chacun des résidus s'ils sont en hélice ou non.

```
[[48.6, 53.4], [-124.9, 156.7], [-66.2, -30.8], [-58.8, -43.1], [-73.9, -40.
[-53.7, -37.5], [-80.6, -16.0], [-68.5, 135.0], [-64.9, -23.5], [-66.9, -45.
[-69.6, -41.0], [-62.7, -37.5], [-68.2, -38.3], [-61.2, -49.1], [-59.7, -41.
[-63.2, -48.5], [-65.5, -38.5], [-64.1, -40.7], [-63.6, -40.8], [-66.4, -44.
[-56.0, -52.5], [-55.4, -44.6], [-58.6, -44.0], [-77.5, -39.1], [-91.7, -11
[48.6, 53.4]]
```

4. Soit `seq` la séquence d'acides aminés suivante : ARAWWAWARWWTAGATWGWTWAWWWATAG. Calculez la fréquence de l'alanine, arginine, tryptophane, thréonine et glycine dans cette séquence.

5. Les étudiants de cette année se sont améliorés, voici leur note :

```
[8, 4, 12, 11, 9, 14, 19, 11, 19, 4, 18, 12, 19, 3, 5]
```

Écrire un script qui retrouve la note maxi, la note mini et qui calcule la moyenne. Améliorer le script en lui faisant évaluer le nombre de mentions : redoublement (< 10), passable (entre 10 et 12), assez-bien (entre 12 et 14) et bien (> 14).

6. Faire une boucle de 0 à 20, afficher les nombres pairs <= 10 d'une part, et afficher les nombres impairs > 10 d'autre part. Pour cet exercice, vous pourrez utiliser l'opérateur modulo % qui retourne le reste de la division entière entre deux nombres.

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un shell.

6 Les fichiers

6.1 Lecture dans un fichier

Dans la plupart des travaux de programmation, on est obligé d'aller lire ou écrire dans un fichier. Si vous avez une petite expérience du C, ce travail est parfois fastidieux surtout lorsque l'on doit extraire une information particulière d'un fichier (*parsing* en anglais). Python possède pour cela tout un tas d'outils qui vous simplifient la vie. Avant de passer à un exemple concret, créez un fichier dans un éditeur de texte que vous sauverez dans votre répertoire avec le nom `exemple.txt`, par exemple :

```
Ceci est la premiere ligne
Ceci est la deuxieme ligne
Ceci est la troisieme ligne
Ceci est la quatrieme et derniere ligne
```

Ensuite, testez cet exemple :

```
>>> filin = open('exemple.txt', 'r')
>>> filin
<open file 'exemple.txt', mode 'r' at 0x40155b20>
>>> filin.readlines()
['Ceci est la premiere ligne\n', 'Ceci est la deuxieme ligne\n',
'Ceci est la troisieme ligne\n', 'Ceci est la quatrieme et derniere ligne\n']
>>> filin.close()
>>> filin
<closed file 'exemple.txt', mode 'r' at 0x40155b20>
>>>
```

- La première commande ouvre le fichier `exemple.txt` en lecture seule (ceci est indiqué avec la lettre `r`). Remarquez que le fichier n'est pas encore lu, mais simplement ouvert (*un peu comme lorsqu'on ouvre un livre, mais qu'on ne l'a pas encore lu*). Lorsqu'on affiche la valeur de la variable `filin`, vous voyez que Python la considère comme un objet de type fichier (`<open file 'exemple.txt', mode 'r' at 0x40155b20>`). Eh oui, Python est un langage orienté **objet**; retenez seulement que l'on peut considérer chaque variable comme un objet, et que l'on peut appliquer des **méthodes** sur chacun de ces objets.
- À propos de méthode, on applique la méthode `readlines()` sur l'objet `filin` dans l'instruction suivante (remarquez la syntaxe du type `objet.méthode`). Ceci nous retourne une liste contenant toutes les lignes du fichier (*dans notre analogie avec un livre, ceci correspondrait à lire les lignes du livre*).
- Enfin, on applique la méthode `close` sur l'objet `filin`, ce qui vous vous en doutez, va fermer le fichier (ceci correspondrait bien sûr à fermer le livre). On pourra remarquer que l'état de l'objet a changé (`<closed file 'exemple.txt', mode 'r' at 0x40155b20>`).

Vous n'avez bien-sûr pas à retenir ces concepts d'objets pour pouvoir programmer avec Python, nous avons juste ouvert cette parenthèse pour attirer votre attention sur la syntaxe (pour ceux qui ont fait du C, ceci ressemble étrangement aux structures, n'est-ce pas ?) Revenons à nos moutons, regardez dans l'exemple suivant en quoi ces opérations nous facilitent la vie :

```
>>> filin = open('exemple.txt', 'r')
>>> lignes = filin.readlines()
>>> lignes
['Ceci est la premiere ligne\n', 'Ceci est la deuxieme ligne\n',
'Ceci est la troisieme ligne\n', 'Ceci est la quatrieme et derniere ligne\n']
```

```
>>> for i in lignes:
...     print i
...
Ceci est la premiere ligne

Ceci est la deuxieme ligne

Ceci est la troisieme ligne

Ceci est la quatrieme et derniere ligne

>>> filin.close()
>>>
```

Vous voyez qu'en 3 ou 4 lignes de code, vous avez lu et parcouru le fichier !

Remarques :

- Notez que la liste `lignes` contient un `\n` à la fin de chacun de ses éléments. Ceci correspond au saut à la ligne de chacune d'entre elles (ceci est codé par un caractère spécial que l'on symbolise par `\n` comme en C). Vous pourrez parfois rencontrer également la notation octale `\012`.
- Vous pouvez aussi remarquer que lorsqu'on affiche les différentes lignes du fichier à l'aide de la boucle et de l'instruction `print`, Python saute à chaque fois une ligne.

Il existe d'autres méthodes que `readlines()` pour lire (et manipuler) un fichier : la méthode `read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique :

```
>>> filin = open("exemple.txt", "r")
>>> filin.read()
'Ceci est la premiere ligne\nCeci est la deuxieme ligne\nCeci est la troisieme l
Ceci est la quatrieme et derniere ligne\n'
>>> filin.close()
```

La méthode `readline()` (sans `s!`) lit un fichier ligne par ligne (on l'utilise donc la plupart du temps avec une boucle `while`), et renvoie la ligne actuellement lue sous forme de chaîne de caractères. Les méthodes `seek` et `tell` permettent respectivement de se déplacer au n^e caractère (plus exactement au n^e octet) d'un fichier et d'afficher où en est la lecture du fichier (quel caractère [ou octet] on est en train de lire).

```
>>> filin = open("exemple.txt", "r")
>>> filin.tell()
0L
>>> filin.readline()
'Ceci est la premiere ligne\n'
>>> filin.tell()
27L
>>> filin.seek(0)
>>> filin.tell()
0L
>>> filin.readline()
'Ceci est la premiere ligne\n'
>>> filin.close()
>>>
```

On remarque que lorsqu'on ouvre un fichier, le tout premier caractère est considéré comme le caractère 0 (tout comme le premier élément d'une liste). La méthode `seek` permet facilement de remonter au début du fichier lorsqu'on est arrivé à la fin où lorsqu'on en a lu une partie.

6.2 Écriture dans un fichier

Écrire dans un fichier est aussi simple que de le lire, et la plupart d'entre vous devineront comment faire... Voyez l'exemple suivant :

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu', 'ornithorynque']
>>> filout = open('exemple2.txt', 'w')
>>> for i in animaux:
...     filout.write(i)
...
>>> filout.close()
>>>
[fuchs@opera ~]$ more exemple2.txt
girafehippopotamesingedahuornithorynque
[fuchs@opera ~]$
```

Quelques commentaires sur cet exemple :

- Après avoir initialisé la liste `animaux`, nous avons ouvert un fichier mais cette fois-ci en mode écriture (avec le caractère `w`).
- Ensuite, on a balayé cette liste à l'aide d'une boucle, lors de laquelle nous avons écrit chaque élément de la liste dans le fichier. Remarquez à nouveau la méthode `write('chaîne de caractères')` qui s'applique sur l'objet `filout`.
- Enfin, on a fermé le fichier avec la méthode `close`. Pour vérifier le contenu du nouveau fichier, nous avons quitté Python et regardé son contenu avec la commande `more`.

Vous voyez qu'il est extrêmement simple en Python de lire ou d'écrire dans un fichier.

Remarque : si votre programme produit uniquement du texte, vous pouvez l'écrire sur la sortie standard (avec l'instruction `print`); l'avantage est que dans ce cas l'utilisateur peut bénéficier de toutes les potentialités d'Unix (redirection, tri, parsing...). S'il veut écrire le résultat du programme dans un fichier, il pourra toujours le faire en redirigeant la sortie.

6.3 Exercices

1. En reprenant l'exemple 'Ceci est la première ligne etc...' ci-dessus, comment expliquer vous que Python saute une ligne à chaque itération ? Réécrivez les instructions *ad-hoc* pour que Python écrive le contenu du fichier sans sauter de ligne.
2. En reprenant le dernier exemple sur l'écriture dans un fichier, vous pouvez constater que les noms d'animaux ont été écrits les uns à la suite des autres, sans retour à la ligne. Comment expliquez-vous ce résultat ? Modifiez les instructions de manière à écrire un animal par ligne.
3. Dans cet exercice, nous allons utiliser une sortie partielle de DSSP (*Define Secondary Structure of Proteins*), qui est un logiciel d'extraction des structures secondaires. Ce fichier contient 5 colonnes correspondant respectivement au numéro de résidu, à l'acide aminé, sa structure secondaire et ses angles phi/psi.
 - Sauvez le fichier disponible [ici](#) dans votre répertoire de travail (jetez-y un petit coup d'oeil en passant...).
 - Chargez les lignes de ce fichier en les plaçant dans une liste ; fermer le fichier.
 - Écrivez chaque ligne à l'écran pour vérifier que vous avez bien chargé le fichier.
 - Écrivez dans un fichier `output.txt` chacune des lignes (on veut des retours à la ligne pour chaque acide aminé).
 - Écrivez dans un fichier `output2.txt` chacune des lignes suivies du message `line checked` (encore une fois, on veut des retours à la ligne pour chaque acide aminé).

4. Télécharger [le fichier PDB \(1BTA\) de la barstar](#). En utilisant Python, calculez le nombre de lignes de ce fichier. Par ailleurs, déterminez de quel organisme vient cette protéine en affichant la ligne 9. Affichez uniquement les lignes correspondant aux carbones alpha (lignes contenant le champ `ATOM` et le nom d'atome `CA`).
5. Exercice pour les GEEKS : Soit le triangle de Pascal :

```
0 1
1 11
2 121
3 1331
4 14641
...
```

À partir de l'ordre 1 (ligne 2), générez l'ordre suivant (on peut utiliser une liste préalablement générée avec `range`). Généralisez à l'aide d'une boucle. Écrivez dans un fichier `pascal.out` les lignes du triangle de Pascal de l'ordre 1 jusqu'à l'ordre 10.

Conseil : pour les exercices 1 à 3, utiliser l'interpréteur Python !

7 Les modules

7.1 Définition

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (l'équivalent en C serait les bibliothèques ou *librairies*). Les développeurs de Python ont mis au point toute une série de modules effectuant une quantité phénoménale de tâches ; c'est la raison pour laquelle je vous conseille de vous renseigner si une partie de code que vous souhaitez écrire n'existe pas déjà. La plupart de ces modules sont déjà installés dans les versions standards de Python, et vous pouvez accéder à [une documentation exhaustive](#) (surfez un peu sur ce site, la quantité de modules est impressionnante).

7.2 Importation de modules

Jusqu'à maintenant, nous avons rencontré une fois cette notion de module lorsque nous avons voulu tirer un nombre aléatoire. Regardons cet exemple à nouveau :

```
>>> import random
>>> random.randint(0,10)
9
>>>
```

Regardons de plus près cet exemple :

- L'instruction `import` vous permet d'importer toutes les fonctions du module `random`
- Ensuite, nous utilisons la fonction (ou méthode) `randint(a,b)` du module `random` ; attention cette fonction renvoie un nombre entier aléatoirement entre `a` inclus et `b` inclus (contrairement à `range` par exemple). Remarquez encore une fois la notation objet `random.randint` (la fonction `randint` pouvant être considérée comme une méthode de l'objet `random`).

Il existe un autre moyen d'importer une ou des fonctions d'un module :

```
>>> from random import randint
>>> randint(0,10)
10
>>>
```

À l'aide du mot-clé `from`, vous pouvez importer une fonction spécifique d'un module donné. Remarquez que dans ce cas il est inutile de répéter le nom du module, seul le nom de ladite fonction est requis.

On peut également importer toutes les fonctions d'un module :

```
>>> from random import *
>>> x = [1,2,3,4,5,6,8,9,10]
>>> shuffle(x)
>>> x
[1, 4, 8, 10, 3, 5, 9, 6, 2]
>>> shuffle(x)
>>> x
[6, 9, 5, 10, 8, 3, 4, 1, 2]
>>> randint(0,100)
52
>>> uniform(0,3.1415926535897931)
1.3634213820695191
>>>
```

Comme vous l'avez deviné, l'instruction `from random import *` importe toutes les fonctions du module `random`. On peut ainsi utiliser toutes ses fonctions, par exemple `shuffle`, qui permute une liste aléatoirement. Enfin, si vous voulez vider de la mémoire un module déjà chargé, vous pouvez utiliser l'instruction `del` :

```
>>> import random
>>> random.randint(0,234)
188
>>> del random
>>> random.randint(0,234)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'random' is not defined
>>>
```

Vous pouvez constater qu'un rappel d'une fonction du module `random` après l'avoir vidé de la mémoire retourne un message d'erreur.

7.3 Obtenir de l'aide sur les modules importés

Pour obtenir de l'aide sur un module rien de plus simple, il suffit d'invoquer la commande `help` :

```
>>> import random
>>> help(random)
...
```

On peut se promener dans l'aide avec les flèches ou les touches `page-up` et `page-down` (comme dans les commandes `man` ou `more` en Unix). Il est aussi possible d'invoquer de l'aide sur une fonction particulière d'un module en la précisant de la manière suivante : `help (random.randint)`. La commande `help` est en fait une commande plus générale permettant d'avoir de l'aide sur n'importe quel objet chargé en mémoire.

```
>>> x=range(2)
>>> help(x)
Help on list object:

class list(object)
| list() -> new list
| list(sequence) -> new list initialized from sequence's items
|
| Methods defined here:
|
| __add__(...)
|     x.__add__(y) <==> x+y
|
| ...
```

Enfin, si on veut connaître en seul coup d'oeil toutes les méthodes ou variables associées à un objet, on peut utiliser la commande `dir` :

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'System'
```

```
'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType', '__all__', '__builtins__',
'__doc__', '__file__', '__name__', '_acos', '_ceil', '_cos', '_e', '_exp', '_hex',
'_inst', '_log', '_pi', '_random', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>>
```

7.4 Modules courants

Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En voici quelques-uns (liste non exhaustive, pour la liste complète, reportez-vous à [la page des modules sur le site de Python](#)) :

- **math** : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- **sys** : passage d'arguments, gestion de l'entrée/sortie standard...
- **os** : dialogue avec le système d'exploitation (e.g. permet de sortir de Python, lancer une commande en shell, puis de revenir à Python).
- **random** : génération de nombres aléatoires.
- **time** : permet d'accéder à l'heure de l'ordinateur et aux fonctions gérant le temps.
- **calendar** : fonctions de calendrier.
- **profile** : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (*profiling* en anglais).
- **urllib** : permet de récupérer des données sur internet depuis python.
- **Tkinter** : interface python avec Tk (permet de créer des objets graphiques ; nécessite d'installer Tk).
- **string** : opérations sur les chaînes de caractères ; à noter que la plupart des fonctions du module `string` sont maintenant **obsolètes** ; il est maintenant plus correct d'utiliser les **méthodes directement associées aux objets de type string** (cf le chapitre suivant sur les chaînes de caractères).
- **re** : gestion des expressions régulières.
- **pickle** : écriture et lecture de structures python (comme les dictionnaires par exemple).
- ...

Je vous conseille vivement d'aller surfer sur les pages de ces modules par curiosité !

Nous verrons plus tard comment créer ses propres modules lorsqu'on est amené souvent à réutiliser ses propres fonctions.

Enfin, il est à noter qu'il existe de nombreux autres modules qui ne sont pas installés de base dans Python mais qui sont de grand intérêt en bioinformatique (au sens large). Citons-en quelques-uns : `numpy` (algèbre linéaire, transformée de Fourier), `biopython` (recherche dans les banques de données biologiques), `rpy` (dialogue R/Python)...

7.5 Exercices

1. Écrivez la racine carrée des nombres de 10 à 20 (module `math`).
2. Calculez le cosinus de $\pi/2$ (module `math`).
3. Retournez la liste des fichiers du répertoire courant (module `os`, documentation à la page *Process Management*).
4. Écrivez les nombres de 1 à 10 avec 1 seconde d'intervalle (module `time`).
5. Générez une séquence de 20 nombres aléatoires, ceux-ci étant des entiers tirés entre 1 et 4.
6. Générez une séquence aléatoire de 20 paires de bases de 2 manières différentes.

7. Déterminez votre jour (lundi, mardi...) de naissance (module calendar).

Conseil : utilisez l'interpréteur Python !

8 Les chaînes de caractères

8.1 Préambule

Nous avons déjà abordé les chaînes de caractères dans le chapitre *variables et écriture* et dans celui des *listes* ; ici nous allons un peu plus loin notamment avec les [méthodes associées aux chaînes de caractères](#). Il est à noter qu'il existe un module `string` mais qui est maintenant considéré comme obsolète.

8.2 Chaînes de caractères et listes

On a déjà vu que les chaînes de caractères pouvaient être considérées comme des listes ; ceci nous permet donc d'utiliser certaines propriétés des listes telles que récupérer des tranches :

```
>>> chaine = "Ceci est une belle chaine de caracteres !"
>>> chaine
'Ceci est une belle chaine de caracteres !'
>>> chaine[0:10]
'Ceci est u'
>>> chaine[15:]
'lle chaine de caracteres !'
>>> chaine[:-4]
'Ceci est une belle chaine de caracter'
>>>
```

A contrario des listes, les chaînes de caractères présentent toutefois une différence notable, elles sont **non modifiables**. Une fois définie, vous ne pouvez plus modifier un de ses éléments ; dans un tel cas Python renvoie un message d'erreur :

```
>>> chaine = "Ceci est une belle chaine de caracteres !"
>>> chaine[15]
'!'
>>> chaine[15] = 'L'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>>
```

Par conséquent, si vous voulez modifier une chaîne, vous êtes obligés d'en construire une nouvelle. Pour cela, n'oubliez pas les opérateurs de concaténation (+) et de duplication (*) (cf chapitre *variables et écriture*), ils peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne.

8.3 Caractères spéciaux

Il existe certains caractères spéciaux comme le `\n` que nous avons déjà vu (pour le retour à la ligne). Le `\t` vous permet d'écrire une tabulation, et si vous voulez écrire un guillemet au sein d'autres guillemets, vous pouvez utiliser `\'` ou `\"` :

```
>>> print "J'imprime un retour a la ligne\npuis une tabulation\t, puis un guille
J'imprime un retour a la ligne
puis une tabulation      , puis un guillemet"
>>> print 'J\'imprime un guillemet simple'
J'imprime un guillemet simple
>>>
```

Lorsqu'on souhaite écrire un texte sur plusieurs lignes, il est très commode d'utiliser les guillemets triples permettant de conserver le formatage (notamment les retours à la ligne) :

```
>>> x = '''J\'aime ecrire ma prose
... sur plusieurs lignes.
... Les guillemets triples sont mes amis
... lors de mes divagations dominicales.
... '''
>>> x
"J\'aime ecrire ma prose\nsur plusieurs lignes.\nLes guillemets triples sont mes
lors de mes divagations dominicales.\n"
>>> print x
J\'aime ecrire ma prose
sur plusieurs lignes.
Les guillemets triples sont mes amis
lors de mes divagations dominicales.

>>>
```

8.4 Les méthodes associées aux objets de type string

Anciennement, on utilisait le module `string` pour accéder aux fonctions de manipulation de chaînes de caractères. Il est maintenant devenu obsolète, on utilise donc plutôt les [méthodes](#) directement associées aux objets de type `string`. Voici quelques exemples :

```
>>> x = 'CECI EST UN TEXTE EN MASJUSCULE'
>>> x.lower()
'ceci est un texte en masjuscule'
>>> x
'CECI EST UN TEXTE EN MASJUSCULE'
>>> 'ceci est un texte en minuscule'.upper()
'CECI EST UN TEXTE EN MINUSCULE'
>>>
```

On voit que les fonctions `lower()` et `upper()` passent un texte en minuscule et majuscule respectivement. On remarque que l'utilisation de ces fonctions n'altèrent pas la chaîne de départ, mais renvoie la chaîne transformée.

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la fonction `split` :

```
>>> ligne = 'Ceci est une ligne'
>>> ligne.split()
['Ceci', 'est', 'une', 'ligne']
>>>
>>> for i in ligne.split():
...     print i
...
Ceci
est
une
ligne
>>>
```

La fonction `split` découpe la ligne en champs, en utilisant comme séparateur les espaces ou les tabulations. Il est possible de modifier le séparateur de champs, par exemple :

```
>>> ligne = 'Cette:ligne:est:separee:par:des:deux-points'
>>> ligne.split(':')
['Cette', 'ligne', 'est', 'separee', 'par', 'des', 'deux-points']
>>>
```

On trouve aussi la fonction `replace`, qui serait l'équivalent de la fonction de substitution du programme unix `sed` :

```
>>> ligne = 'Ceci est une ligne'
>>> ligne.replace('Ceci', 'Cela')
'Cela est une ligne'
>>>
```

J'espère qu'après ce petit tour d'horizon vous serez convaincu de la richesse des méthodes associées aux chaînes de caractères ! Pour avoir une liste exhaustive de l'ensemble de ces méthodes, vous pouvez utiliser la commande `dir` (pour l'instant vous pouvez ignorer les méthodes qui commencent et qui se terminent par 2 underscores `'_'`). Vous pouvez ensuite accéder à l'aide et à la documentation d'une fonction particulière avec `help` :

```
>>> dir(ligne)
['_add_', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__neq__',
 '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
 '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__radd__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
 'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
 'zfill']
>>> help(ligne.split)
Help on built-in function split:

split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done.  If sep is not specified or is None, any
    whitespace string is a separator.

(END)
```

8.5 Conversion de types

Dans tout langage de programmation, on est souvent amené à convertir les types, i.e. passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int`, `float` et `str`; regardez l'exemple suivant :

```
>>> i = 3
>>> str(i)
'3'
>>> i = '456'
>>> int(i)
456
```

```
>>> float(i)
456.0
>>> i = '3.1416'
>>> float(i)
3.1415999999999999
>>>
```

Ces conversions sont essentielles lorsqu'on lit ou écrit des nombres dans un fichier. Par exemple, les nombres dans un fichier sont considérés comme du texte par la fonction `readlines()`, par conséquent il faut les convertir si on veut effectuer des opérations numériques dessus.

8.6 Exercices

1. Soit la séquence suivante en acides aminés : ALA GLY GLU ARG TRP TYR SER GLY ALA TRP. Transformez cette séquence en une chaîne de caractères en utilisant le code 1 lettre pour les acides-aminés.
2. Téléchargez le [fichier pdb 1BTA](#) sur le disque. Faites un script qui récupère seulement les carbones alpha et qui les affiche à l'écran.
3. En se basant sur le script précédent, affichez à l'écran les carbones alpha des 2 premiers résidus. Toujours avec le même script, calculez la distance inter atomique entre ces 2 atomes.
4. En se basant sur le script précédent, calculez les distances entre Carbones alpha consécutifs. Affichez ces distances avec à la fin la moyenne de celles-ci.
5. Appliquez le même script au [fichier PDB 1WXR](#). Le script fonctionne-t-il toujours (sachant que la distance inter carbone alpha dans les protéines est très stable = 3.8 angströms) ? Y a-t-il certaines précautions à prendre avec ce nouveau fichier ?

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un shell

9 Plus sur les listes

9.1 Propriétés des listes

Jusqu'à maintenant, nous avons toujours utilisé des listes qui étaient déjà remplies. Nous savons comment modifier un de ses éléments, mais il est aussi parfois très pratique d'en remanier la taille (e.g. ajouter, insérer ou supprimer un ou plusieurs éléments etc...). Les listes possèdent pour ça des méthodes qui leurs sont propres comme `append`, `insert`, `sort`, `reverse` et `del`. Regardez les exemples suivants :

```
- liste.append(element) : ajoute element à la fin de la liste
>>> x = [1,2,3]
>>> x.append(2239)
>>> x
[1, 2, 3, 2239]
- del liste[indice] : supprime element à la position indice de la liste
>>> del x[1]
>>> x
[1, 3, 2239]
- liste.insert(indice, objet) : insère un objet dans une liste avant l'indice
>>> x.insert(2,-15)
>>> x
[1, 3, -15, 2239]
- liste.sort() : tri la liste
>>> x.sort()
>>> x
[-15, 1, 3, 2239]
- liste.reverse() : inverse la liste
>>> x.reverse()
>>> x
[2239, 3, 1, -15]
>>>
```

Remarque : attention, une liste remaniée n'est pas renvoyée ! Pensez-y dans vos utilisations futures des listes.

La méthode `append` est particulièrement pratique car elle permet de construire une liste associée à une boucle ; pour cela il est commode de définir préalablement une liste vide de la forme `liste = []`. Par exemple, si on veut convertir une chaîne de caractères en liste on peut utiliser quelque chose de la forme :

```
>>> seq = 'CAAAGGTAACGC'
>>> seq_list = []
>>> seq_list
[]
>>> for i in seq:
...     seq_list.append(i)
...
>>> seq_list
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
>>>
```

Remarquez que l'on peut utiliser directement la fonction `list(sequence)` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, tuples, etc) et qui renvoie une liste :

```
>>> seq = 'CAAAGGTAACGC'
```

```
>>> list(seq)
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
>>>
```

Cette méthode est certes plus simple, mais il arrive parfois que l'on doive utiliser les boucles tout de même, e.g. lorsqu'on lit un fichier. On peut aussi utiliser les opérateurs de concaténation et de duplication (+ et *) comme pour les chaînes de caractères :

```
>>> liste = ['a', 'b', 'c', 'd', 'e', 'f']
>>> liste + ['g', 'h']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> liste * 2
['a', 'b', 'c', 'd', 'e', 'f', 'a', 'b', 'c', 'd', 'e', 'f']
>>>
```

9.2 Références partagées

Il est très important de savoir que l'affectation d'une liste (à partir d'une liste préexistante) crée en réalité une **référence** et non une **copie** :

```
>>> x = [1, 2, 3]
>>> y = x
>>> y
[1, 2, 3]
>>> x[1] = -15
>>> y
[1, -15, 3]
>>>
```

Vous voyez que la modification de `x` modifie `y` aussi. Rappelez-vous de ceci dans vos futurs programmes car cela pourrait avoir des effets désastreux ! Techniquement, Python utilise des pointeurs (comme en C) vers les mêmes objets et ne crée pas de copie à moins que vous n'en ayez fait la demande explicitement. Regardez cet exemple :

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x[1] = -15
>>> y
[1, 2, 3]
>>>
```

Vous pouvez utiliser aussi la fonction `list` qui renvoie une liste explicitement :

```
>>> x = [1, 2, 3]
>>> y = list(x)
>>> x[1] = -15
>>> y
[1, 2, 3]
>>>
```

9.3 Exercices

1. Générer une séquence aléatoire de 20 paires de base en utilisant une liste et la méthode `append`.

2. Transformer la séquence en séquence complémentaire inverse (n'oubliez pas que la séquence complémentaire doit être inversée, pensez aux méthodes des listes!).
3. Générer une séquence aléatoire de 50 bases contenant 10 % de A, 50 % de G, 30 % de T et 10 % de C.

10 Les dictionnaires et les tuples

10.1 Les Dictionnaires

Les **dictionnaires** constituent un type intégré de Python qui se révèle très pratique lorsque l'on a des structures complexes à décrire, et que les listes présentent leurs limites. Ils sont des collections non ordonnées d'objets, c-à-d qu'il n'y a pas de notion de "déplacement" (i.e. par des indices). On accède aux **valeurs** d'un dictionnaire par des **clés**; ceci semble un peu confus? Regardez l'exemple suivant :

```
>>> disque1 = {}
>>> disque1['year']=2000
>>> disque1['style'] = 'Trash Metal'
>>> disque1['name'] = 'Far beyond driven'
>>> disque1['author'] = 'Pantera'
>>> disque1
{'year': 2000, 'name': 'Far beyond driven', 'author': 'Pantera', 'style': 'Trash
>>> disque1['author']
'Pantera'
>>>
```

En premier, on a défini un dictionnaire vide avec les symboles {} (tout comme on peut le faire pour les listes avec []). Ensuite, on a rempli le dictionnaire avec différentes clés auxquelles on a affecté des valeurs (une par clé); en fait vous pouvez remarquer que l'on peut mettre autant de clés que l'on veut dans un dictionnaire (tout comme on peut ajouter autant d'éléments que voulu dans une liste). Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style `dictionnaire['cle']`.

Les méthodes `keys` et `values` renvoient comme vous vous en doutez les clés et les valeurs d'un dictionnaire (sous forme de liste) :

```
>>> disque1.keys()
['year', 'name', 'author', 'style']
>>> disque1.values()
[2000, 'Far beyond driven', 'Pantera', 'Trash Metal']
>>>
```

On peut aussi initialiser toutes les clés d'un dictionnaire en une seule opération :

```
>>> disque2 = {'year': 1994, 'name': 'Concerto N2', 'author': 'Rachmaninov',
'style': 'classic'}
>>>
```

En créant une liste de dictionnaires possédant les mêmes clés, on accède à leur intérêt principal (qui pourrait rappeler les bases de données) :

```
>>> discotheque = [disque1,disque2]
>>> discotheque
[{'year': 2000, 'name': 'Far beyond driven', 'author': 'Pantera', 'style': 'Tras
{'year': 1994, 'name': 'Concerto N2', 'author': 'Rachmaninov', 'style': 'classic
>>> for i in discotheque:
...     print i['author']
...
Pantera
Rachmaninov
>>>
```

Enfin, pour vérifier si une clé existe, vous pouvez utiliser la propriété `has_key()` :

```
>>> if disque2.has_key('style') :
...     print "le champ 'style' existe dans le dictionnaire disque2"
...
le champ 'style' existe dans le dictionnaire disque2
>>>
```

Vous voyez que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes (en C, cela pourrait correspondre aux structures).

10.2 Les Tuples

Les **tuples** correspondent aux listes à la différence qu'ils sont **non modifiables**. On a vu à la section précédente que les listes pouvaient être modifiées par des références ; les tuples vous permettent de vous affranchir de ce problème. Pratiquement, ils utilisent les parenthèses au lieu des crochets :

```
>>> x = (1, 2, 3)
>>> x
(1, 2, 3)
>>> x[2]
3
>>> x[0:2]
(1, 2)
>>> x[2] = 15
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>>
```

L'affectation et l'indilage fonctionne comme avec les listes, mais si l'on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un autre tuple :

```
>>> x = (1, 2, 3)
>>> x + (2,)
(1, 2, 3, 2)
>>>
```

Remarquez que lorsque l'on veut utiliser un tuple d'un seul élément, on doit utiliser une syntaxe avec une virgule : `(element,)`, ceci pour éviter une ambiguïté avec une simple expression. Autre particularité des tuples, il est possible d'en créer de nouveaux sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression :

```
>>> x = (1, 2, 3)
>>> x
(1, 2, 3)
>>> x = 1, 2, 3
>>> x
(1, 2, 3)
>>>
```

Toutefois, je vous conseille dans un premier temps d'utiliser systématiquement les parenthèses afin d'éviter les confusions !

Enfin, on peut utiliser la fonction `tuple` (sequence) qui fonctionne exactement comme la fonction `list`, c-à-d qu'elle prend en argument un objet séquentiel et renvoie le tuple correspondant :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple("ATGCCGCGAT")
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
```

Remarque : les listes, dictionnaires, tuples sont des objets qui peuvent contenir des collections d'autres objets. On peut donc construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples ou des listes etc...

10.3 Exercices

1. À partir du [fichier PDB 1BTA](#), construisez un dictionnaire qui contient 4 clés se référant au premier carbone alpha : le numéro du résidu, puis les coordonnées x, y et z.
2. Sur le même modèle que ci-dessus, créez une liste de dictionnaires pour chacun des carbones alpha de la protéine.
3. À l'aide de cette liste, calculez les coordonnées x, y et z du barycentre de ces carbones alpha.
4. Soit la séquence nucléotidique suivante :

```
ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG
```

Faites un programme qui répertorie tous les mots de 2 lettres qui existent dans la séquence (AA, AC, AG, AT, etc) ainsi que leur nombre d'occurrences (utiliser pour cela un dictionnaire et ses propriétés), et qui les affiche à l'écran.

5. Faites de même avec des mots de 3 et 4 lettres.
6. En vous basant sur les scripts précédents, extrayez les mots de 2 lettres et leur occurrence sur le génome du virus de l'herpes.
7. Créez un script `extract-words.py` qui prend en arguments un fichier genbank suivi d'un entier compris entre 1 et 4. Ce script doit extraire du fichier genbank tous les mots (ainsi que leur nombre d'occurrences) du nombre de lettres passées en option.
8. Appliquez ce script sur le génome d'*Escherichia coli* : [NC_000913.gbk](#). Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*E. Coli*? Comment pourrait-on en améliorer la rapidité?

11 Les fonctions

11.1 Définition

Les fonctions représentent un élément essentiel en programmation, lorsque l'on veut réaliser plusieurs fois la même opération au sein d'un même programme. Non seulement cela vous évite de réécrire plusieurs fois le même code, mais en outre, cela augmente considérablement la lisibilité. Il est en effet plus judicieux d'écrire un programme contenant plusieurs petites fonctions simples. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité comparées à celles d'un langage de bas niveau comme C. Pour définir une fonction, Python utilise le mot-clé `def`, et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé `return`. Par exemple :

```
>>> def carre(x):
...     return x**2
...
>>> print carre(2)
4
>>>
```

Remarquez également que la syntaxe de `def` utilise les `:` comme les boucles `for`, `while` ainsi que les tests `if`. De même que pour les boucles et les tests, l'**indentation** du corps de la fonction est **obligatoire**.

Dans cet exemple, on a passé un argument à la fonction `carre()`, et elle nous a retourné une valeur que nous avons affichée à l'écran. Que veut dire valeur retournée ? Et bien ça signifie que cette dernière est stockable dans une variable :

```
>>> res = carre(2)
>>> print res
4
>>>
```

Ici nous avons stocké le résultat renvoyé par la fonction dans une variable `res`. Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur, par exemple :

```
>>> def hello():
...     print "hello"
...
>>> hello()
hello
>>>
```

Dans ce cas la fonction `hello()` se contente d'imprimer la chaîne "hello" à l'écran et ne renvoie aucun résultat. Par conséquent, cela n'a pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction. Si on essaie tout de même (certaines personnes ne croient que ce qu'elles voient ;-)), Python affecte la valeur 'None' qui signifie 'Rien' en anglais :

```
>>> x = hello()
hello
>>> print x
None
>>>
```

11.2 Passage d'arguments

Le nombre d'argument(s) que l'on peut passer à une fonction est variable à souhait ; nous avons vu ci-dessus des fonctions auxquelles on passait 0 ou 1 argument. Dans les leçons précédentes, on a par ailleurs vu des fonctions internes à Python qui prenaient au moins 2 arguments, pour rappel souvenez-vous de `range(1, 10)` ou encore `range(1, 10, 2)`. Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui est en train de développer une nouvelle fonction.

Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au *typage dynamique*, c'est à dire qu'il reconnaît pour vous le type des variables au moment de l'exécution, par exemple :

```
>>> def fois(x,y):
...     return x*y
...
>>> fois(2,3)
6
>>> fois(3.1415,5.23)
16.430045000000003
>>> fois('toto',2)
'toto'
>>>
```

L'opérateur `*` reconnaissant plusieurs types (entiers, réels, chaînes de caractères), notre fonction est capable d'effectuer plusieurs tâches ! Un autre gros avantage de Python est que ses fonctions sont capables de renvoyer plusieurs valeurs à la fois (ce qui est faisable mais moins évident en C!). Regardez cette fraction de code :

```
>>> def carre-et-cube(x):
...     return x**2,x**3
...
>>> carre-et-cube(2)
(4, 8)
>>>
```

Vous voyez qu'en réalité Python renvoie un objet séquentiel, qui peut par conséquent contenir plusieurs valeurs. Dans notre exemple Python renvoie un objet `tuple` car on a utilisé une syntaxe de ce type (on a vu à la section précédente que plusieurs variables (ou constantes) séparées par des virgules représentaient un `tuple`). Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre-et-cube(x):
...     return [x**2,x**3]
...
>>> carre-et-cube(2)
[4, 8]
>>>
```

Enfin, il est possible de passer un ou plusieurs argument(s) de manière facultative et de leur attribuer une valeur par défaut :

```
>>> def useless_fct(x=1):
...     return x
```

```
...
>>> useless_fct ()
1
>>> useless_fct (10)
10
>>>
```

À noter, si on passe plusieurs arguments à une fonction, le ou les arguments facultatifs doivent être situés à droite (càd `def fct(x, y, z=1) :` et non pas `def fct(z=1, x, y) :`).

11.3 Portée des variables

Il est très important lorsqu'on manipule des fonctions de connaître la portée des variables. Premièrement, on peut créer des variables au sein d'une fonction qui ne seront pas visibles à l'extérieur de celle-ci ; on les appelle **variables locales**. Regardez le code suivant :

```
>>> def mafonction():
...     x = 2
...     print 'x vaut ',x,'dans la fonction'
...
>>> mafonction()
x vaut 2 dans la fonction
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
>>>
```

Lorsque Python exécute le code de la fonction, il connaît le contenu de la variable `x` ; par contre, lorsqu'on est revenu dans le module principal (dans notre cas, il s'agit de l'interpréteur avec l'invite `'>>> '`), il ne la connaît plus d'où le message d'erreur. De même, une variable passée en argument est considérée comme **locale** lorsqu'on arrive dans la fonction :

```
>>> def mafonction(x):
...     print 'x vaut ',x,'dans la fonction'
...
>>> mafonction(2)
x vaut 2 dans la fonction
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
>>>
```

Deuxièmement, une variable déclarée à la racine du module (c'est comme cela que l'on appelle un programme Python), càd dans la fonction principale (équivalent au `main()` du C), est dite **variable globale** ; une telle variable est visible dans tout le module :

```
>>> def mafonction():
...     print x
...
>>> x = 3
>>> mafonction()
3
```

```
>>> print x
3
>>>
```

Dans ce cas, la variable `x` est visible dans le module principal et dans toutes les fonctions du module. Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction :

```
>>> def fct():
...     x += 1
...
>>> x=1
>>> fct()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fct
UnboundLocalError: local variable 'x' referenced before assignment
>>>
```

L'erreur renvoyée montre que Python pense que `x` est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé `global` :

```
>>> def fct():
...     global x
...     x += 1
...
>>> x=1
>>> fct()
>>> x
2
>>>
```

Dans ce dernier cas, le mot-clé `global` a forcé la variable `x` à être globale plutôt que locale au sein de la fonction.

11.4 Portée des listes

J'attire maintenant votre attention sur un point important. Soyez extrêmement attentifs avec les types modifiables (tels que les listes) car vous pouvez les changer au sein d'une fonction :

```
>>> def mafonction():
...     liste[1] = -127
...
>>> liste = [1,2,3]
>>> mafonction()
>>> liste
[1, -127, 3]
>>>
```

De même que si vous passez une liste en argument, elle est tout autant modifiable au sein de la fonction :

```
>>> def mafonction(x):
...     x[1] = -15
```

```

...
>>> y = [1,2,3]
>>> mafonction(y)
>>> y
[1, -15, 3]
>>>

```

Si vous voulez éviter ce problème, utilisez des tuples, Python renverra une erreur puisque ces derniers sont non modifiables ! Une autre solution pour éviter la modification d'une liste lorsqu'elle est passée en tant qu'argument, est de la passer explicitement (comme nous l'avons fait pour l'affectation) afin qu'elle reste intacte dans le programme principal :

```

>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y[:])
>>> y
[1, 2, 3]
>>> mafonction(list(y))
>>> y
[1, 2, 3]

```

Dans ces deux derniers exemples, une copie de `y` est créée à la volée lorsqu'on appelle la fonction, ainsi la liste `y` du module principal reste intacte.

11.5 La règle LGI

Lorsque Python rencontre une variable, il va traiter la résolution de son nom avec des priorités particulières : d'abord il va regarder si la variable est **locale**, puis si elle n'existe pas localement, il vérifiera si elle est **globale** et enfin si elle n'est pas globale, il testera si elle est **interne** (par exemple la fonction `len()` est considérée comme une fonction interne à Python, i.e. elle existe à chaque fois que vous lancez Python). On appelle cette règle la règle **LGI** pour locale, globale, interne. En voici un exemple :

```

>>> def mafonction():
...     x = 4
...     print 'Dans la fonction x vaut ',x
...
>>> x = -15
>>> mafonction()
Dans la fonction x vaut 4
>>> print 'Dans le module principal, x vaut',x
Dans le module principal, x vaut -15
>>>

```

Vous voyez que dans la fonction, `x` a pris la valeur qui lui était définie localement en priorité sur sa valeur définie dans le module principal.

Conseil : même si Python peut reconnaître une variable ayant le même nom que ses fonctions ou variables internes, évitez de les utiliser car ceci rendra votre code confus !

11.6 Exercices

1. Créez une fonction qui prend une liste de bases et qui renvoie la séquence complémentaire d'une séquence d'ADN sous forme de liste.
2. À partir d'une séquence d'ADN `ATCGATCGATCGCTGCTAGC`, renvoyez le brin complémentaire (n'oubliez pas que la séquence doit être inversée).
3. Créez une fonction *distance* qui calcule une distance euclidienne en 3 dimensions. En reprenant le code de l'exercice 3 du chapitre sur les *chaînes de caractères*, refaire le même code en utilisant votre fonction *distance*.

12 Les modules sys et re

12.1 Le module sys : passage d'arguments

Le [module sys](#) contient (comme son nom l'indique) des fonctions et des variables spécifiques au système, ou plus exactement à l'interpréteur lui-même. Par exemple, il permet de gérer l'entrée et la sortie standard (*stdin* et *stdout*). Ce module va particulièrement nous intéresser pour récupérer les arguments passés à la ligne de commande au script python. Dans cet exemple, oublions l'interpréteur, et écrivons le script suivant que l'on enregistrera sous le nom `test.py` (n'oubliez pas de le rendre exécutable !):

```
#!/usr/bin/python

import sys

print sys.argv
```

Ensuite lancez `test.py` suivi de plusieurs arguments, par exemple (ici, `[fuchs@opera ~]$` représente l'invite du shell) :

```
[fuchs@opera ~]$ ./test.py gogo toto 45
['./test.py', 'gogo', 'toto', '45']
[fuchs@opera ~]$
```

`sys.argv` est en fait une liste qui représente tous les arguments de la ligne de commande, y compris le nom du script lui-même. On peut donc accéder à chacun de ces arguments avec `sys.argv[1]`, `sys.argv[2]`...

On peut aussi utiliser la fonction `sys.exit()` pour quitter le script Python. On peut donner comme argument un objet (en général une chaîne de caractères) qui sera renvoyé au moment où Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

```
#!/usr/bin/python

import sys

if len(sys.argv) < 2:
    sys.exit("Usage : test.py file.gbk")

#
# ici commence le script
#
```

Puis on l'exécute sans argument :

```
[fuchs@opera ~]$ ./test.py
Usage : test.py file.gbk
[fuchs@opera ~]$
```

12.2 Le module re : expressions régulières

Le [module re](#) vous permet d'utiliser des expressions régulières au sein de Python. Cet outil puissant est complètement incontournable en bioinformatique lorsqu'on veut aller récupérer des informations dans un fichier. Dans ce cours, nous supposons que vous connaissez déjà la

syntaxe des expressions régulières ; pour de l'aide sur cette syntaxe, on pourra se référer à la [page d'aide des expressions régulières](#) sur le site officiel de Python.

La fonction `search` vous permet de rechercher un motif (*pattern*) au sein d'une chaîne de caractères avec une syntaxe de la forme `search(pattern, string)`. Si `pattern` existe dans `string`, Python renvoie une instance `MatchObject` ; sans rentrer dans les détails propre au langage orienté objet, si on utilise cette instance dans un test, il sera considéré comme vrai. Regardez cet exemple :

```
>>> import re
>>> chaine = "Voici l'histoire de toto !"
>>> re.search('toto', chaine)
<_sre.SRE_Match object at 0x40365218>
>>> if re.search('toto', chaine):
...     print "L'expression reguliere fonctionne !!!"
...
L'expression reguliere fonctionne !!!
>>>
```

Il est à noter qu'il existe aussi la fonction `match()` dans le module `re` qui fonctionne sur le modèle de `search()` ; la différence est qu'elle renvoie une instance `MatchObject` seulement lorsque l'expression régulière *matche* le début de la chaîne (à partir du premier caractère).

```
>>> chaine = "Voici l'histoire de toto !"
>>> re.search('toto', chaine)
<_sre.SRE_Match object at 0x7ff444f0>
>>> re.match('toto', chaine)
>>>
```

Je recommande plutôt l'usage de la fonction `search()` ; si on souhaite *matcher* au début de la chaîne, on peut toujours utiliser l'accroche de début de ligne : `'^'`.

Il est aussi commode de préalablement compiler l'expression régulière à l'aide de la fonction `compile` qui renvoie un objet de type expression régulière :

```
>>> maregex = re.compile('^toto')
>>> maregex
<_sre.SRE_Pattern object at 0x7ff32cf0>
```

On peut alors utiliser directement cet objet avec la méthode `search()` :

```
>>> chaine="toto est le heros de l'histoire"
>>> maregex.search(chaine)
<_sre.SRE_Match object at 0x7ff44410>
>>> chaine="Le heros de l'histoire est toto"
>>> maregex.search(chaine)
>>>
```

Vous vous posez la question, pourquoi Python renvoie une instance `MatchObject` lorsqu'une expression régulière *matche* une chaîne ? Et bien parce qu'on peut réutiliser cet objet pour récupérer des informations sur les zones qui ont *matchées* :

```
>>> regex = re.compile('([0-9+])\.([0-9+])')
>>> MATCH = regex.search("pi : 3.1415926535897931")
>>> MATCH.group(0)
'3.1415926535897931'
>>> MATCH.group(1)
```

```
'3'
>>> MATCH.group(2)
'1415926535897931'
>>> MATCH.start()
5
>>> MATCH.end()
23
>>>
```

On s'aperçoit sur cet exemple que l'utilisation de la fonction `group()` sur une instance `MatchObject` permet de récupérer les zones situées entre parenthèses. Les fonctions `start()` et `end()` donnent respectivement la position de début et de fin de la zone qui *matche* la regex. Il est important de noter que la fonction `search()` ne renvoie que la première zone qui *matche*, même s'il en existe plusieurs :

```
>>> MATCH = regex.search("pi : 3.1415926535897931 , et e : 2.7182818284590451")
>>> MATCH.group(0)
'3.1415926535897931'
```

Si on veut récupérer chaque zone, on peut utiliser la fonction `findall` qui renvoie un tuple avec l'ensemble de ces zones :

```
>>> MATCH = regex.findall("pi : 3.1415926535897931 , et e : 2.7182818284590451")
>>> MATCH
[('3', '1415926535897931'), ('2', '7182818284590451')]
>>>
```

Enfin, la fonction `sub(repl, string)` permet d'effectuer des remplacements assez puissants. Par défaut la fonction `sub` remplace toutes les occurrences de la regex ; si on ne souhaite que les *n* premières occurrences, on peut utiliser l'argument `count=n` :

```
>>> regex.sub('3.14', "pi : 3.1415926535897931 , et e : 2.7182818284590451")
'pi : 3.14 , et e : 3.14'
>>> regex.sub('3.14', "pi : 3.1415926535897931 , et e : 2.7182818284590451", count=1)
'pi : 3.14 , et e : 2.7182818284590451'
>>>
```

Je pense vous avoir démontré la puissance de ce module. Alors, à vos expressions régulières !

12.3 Exercices

Bouquet final : extraction des gènes d'un fichier genbank.

1. Reprogrammez une version simple de `grep` en Python (sans toutes les options) en vous aidant du module `re`. Le script prend comme premier argument l'expression à rechercher, et comme deuxième argument le fichier à analyser. Testez le script sur un fichier genbank en extrayant l'organisme et le locus.
2. À partir d'un fichier genbank (par exemple celui d'*Escherichia coli* : [NC_000913.gbk](#)), et du module `re`, récupérez toutes les lignes qui indiquent l'emplacement du début et de la fin des gènes, du type :

```
gene                58..272
```

3. Faire de même avec les gènes complémentaires :

```
gene                complement(55979..56935)
```

4. À l'aide du module d'expression régulière récupérez la séquence d'ADN et affichez la à l'écran (*vous pouvez utiliser une expression régulière avec `egrep`*).
5. Mettez cette séquence dans une liste, et récupérez les deux premiers gènes (en les affichant à l'écran).
6. À partir de toutes ces petites opérations concevez un programme `genbank2genes.py` qui extrait tous les gènes d'un fichier genbank et les affiche à l'écran. Pour cela vous pourrez utiliser tout ce que nous avons vu jusque maintenant (fonctions, listes, utilisation de modules...)
7. À partir du script précédent, refaites le même programme en écrivant chacun des gènes au format fasta et en le sauvant dans un fichier. Vous utiliserez comme ligne de commentaire, le nom de l'organisme, suivi du numero du gène, du début et de la fin du gène. Les noms de fichier seront de la forme `gene1.fasta`, `gene2.fasta`, etc.

Pour ces exercices, écrivez des scripts dans des fichiers puis exécutez-les dans un shell.

Bon courage !

13 Le module pickle

Le module `pickle` permet d'effectuer une sérialisation et désérialisation des données. Il s'agit en fait d'encoder (d'une manière particulière) les objets que l'on peut créer en Python (variables, listes, dictionnaires, fonctions, classes, *etc*) afin de les stocker dans un fichier, puis de les récupérer, sans devoir effectuer une étape de *parsing*.

13.1 Codage des données

Pour encoder des données avec `pickle` et les envoyer dans un fichier, on peut utiliser la fonction `dump(obj, file)` (où `file` est un fichier déjà ouvert) :

```
>>> import pickle
>>> fileout = open("mydata.dat", "w")
>>> maliste = range(10)
>>> mondico = {'year': 2000, 'name': 'Far beyond driven', 'author': 'Pantera',
              'style': 'Trash Metal'}
>>> pickle.dump(maliste, fileout)
>>> pickle.dump(mondico, fileout)
>>> pickle.dump(3.14, fileout)
>>> fileout.close()
```

Si on observe le fichier créé, on s'aperçoit que les données sont codées :

```
>>> import os
>>> os.system("cat mydata.dat")
(lp0
I0
aI1
aI2
aI3
aI4
aI5
aI6
aI7
aI8
aI9
a.(dp0
S'style'
p1
S'Trash Metal'
p2
sS'author'
p3
S'Pantera'
p4
sS'name'
p5
S'Far beyond driven'
p6
sS'year'
p7
I2000
```

```
s.F3.14000000000000001
.0
>>>
```

13.2 Décodage des données

Pour décoder les données, rien de plus simple, il suffit d'utiliser la fonction `load(file)` où `file` est un fichier ouvert en lecture :

```
>>> filein = open("mydata.dat")
>>> pickle.load(filein)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> pickle.load(filein)
{'style': 'Trash Metal', 'year': 2000, 'name': 'Far beyond driven', 'author': 'P
>>> pickle.load(filein)
3.14000000000000001
```

Attention à ne pas utiliser la fonction `load` une fois de trop, sinon une erreur est générée :

```
>>> pickle.load(filein)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.5/pickle.py", line 1370, in load
    return Unpickler(file).load()
  File "/usr/lib/python2.5/pickle.py", line 858, in load
    dispatch[key](self)
  File "/usr/lib/python2.5/pickle.py", line 880, in load_eof
    raise EOFError
EOFError
>>>
```

Il est à noter qu'il existe un module équivalent mais beaucoup plus rapide, le module [cpi-ckle](#).

13.3 Exercices

1. Reprenez le dictionnaire de mots de 4 lettres du génome du virus de l'herpès (*exercice 7 sur les dictionnaires et tuples*). Encoder puis décoder ce dictionnaire dans un fichier avec `pickle`.

14 Gestion des entrées et des erreurs

Désolé, cette section est en construction !

15 Création de ses propres modules

15.1 Création

Créer ses propres modules est très simple en Python. Il suffit d'écrire un ensemble de fonctions (et/ou de variables) dans un fichier, puis enregistrer celui-ci avec une extension `.py`. À titre d'exemple, voici le contenu du module `message.py`.

```
"""Module inutile qui affiche des messages :-)
"""

def bonjour(nom):
    """Affiche bonjour !
    """
    return "bonjour" + nom

def ciao(nom):
    """Affiche Ciao !
    """
    return "ciao" + nom

def Yo(nom):
    """Affiche Yo !
    """
    return "Yo " + nom + " !"

date=16092008
```

Pour appeler une fonction ou une variable de ce module, il suffit que le fichier `message.py` soit dans le répertoire courant (dans lequel on travaille), ou bien dans un répertoire indiqué par la variable d'environnement Unix `PYTHONPATH`. Ensuite il suffit d'importer le module, et toutes ses fonctions (et variables) vous sont alors accessibles :

```
>>> import message
>>> import os
>>> os.system("ls -l message.py*")
-rw-r--r-- 1 fuchs dsimb 269 Sep 16 23:41 message.py
-rw-r--r-- 1 fuchs dsimb 619 Sep 16 23:45 message.pyc
0
>>>
```

La première fois qu'on importe le module, Python a créé un fichier avec une extension `.pyc` qui contient le [bytecode](#) du module. Notez bien que le fichier est enregistré avec une extension `.py` et pourtant on ne la précise pas lorsqu'on importe le module (`import message`). Ensuite on peut utiliser les fonctions comme avec un module classique :

```
>>> message.Yo("Joe")
'Yo Joe !'
>>> message.ciao("Bill")
'ciaoBill'
>>> message.bonjour("Monsieur")
'bonjourMonsieur'
>>> message.date
16092008
>>>
```

Les commentaires (entre triple guillemets) situés en début de module et sous chaque fonction permettent de fournir de l'aide invoquée ensuite par la commande `help` :

```
>>> help(message)
NAME
    message - Module inutile qui affiche des messages :-)

FILE
    /cygdrive/c/Documents and Settings/pat/python/web/message.py

FUNCTIONS
    Yo(nom)
        Affiche Yo !

    bonjour(nom)
        Affiche bonjour !

    ciao(nom)
        Affiche Ciao !

DATA
    date = 16092008
```

Remarques :

- Les fonctions dans un module peuvent s'appeler les unes les autres.
- Les fonctions dans un module peuvent appeler des fonctions situées dans un autre module s'il a été préalablement importé avec la commande `import`.

Vous voyez que les modules sont d'une simplicité enfantine à créer. Si vous avez des fonctions que vous serez amenés à utiliser souvent, n'hésitez plus !

15.2 Exercices

1. Reprenez l'ensemble des fonctions qui gèrent le traitement de séquence d'acides nucléiques et incluez les dans un module `adn.py`. Testez les au fur et à mesure.